CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- HW6 posted. Will be due on this Sunday midnight.
- Quiz 2 on this Wednesday
- Midterm_Exam1 will be held on:
  - Time: Oct 23rd (Wednesday)
  - Location: the same classroom
  - Topics:
    - The 1st week to this week's topics

Recap

- Abstract data types and Interface
- Sorted linked list
- Recursion

Learning Topics

- Merge Sort
- Quick Sort

Recursion

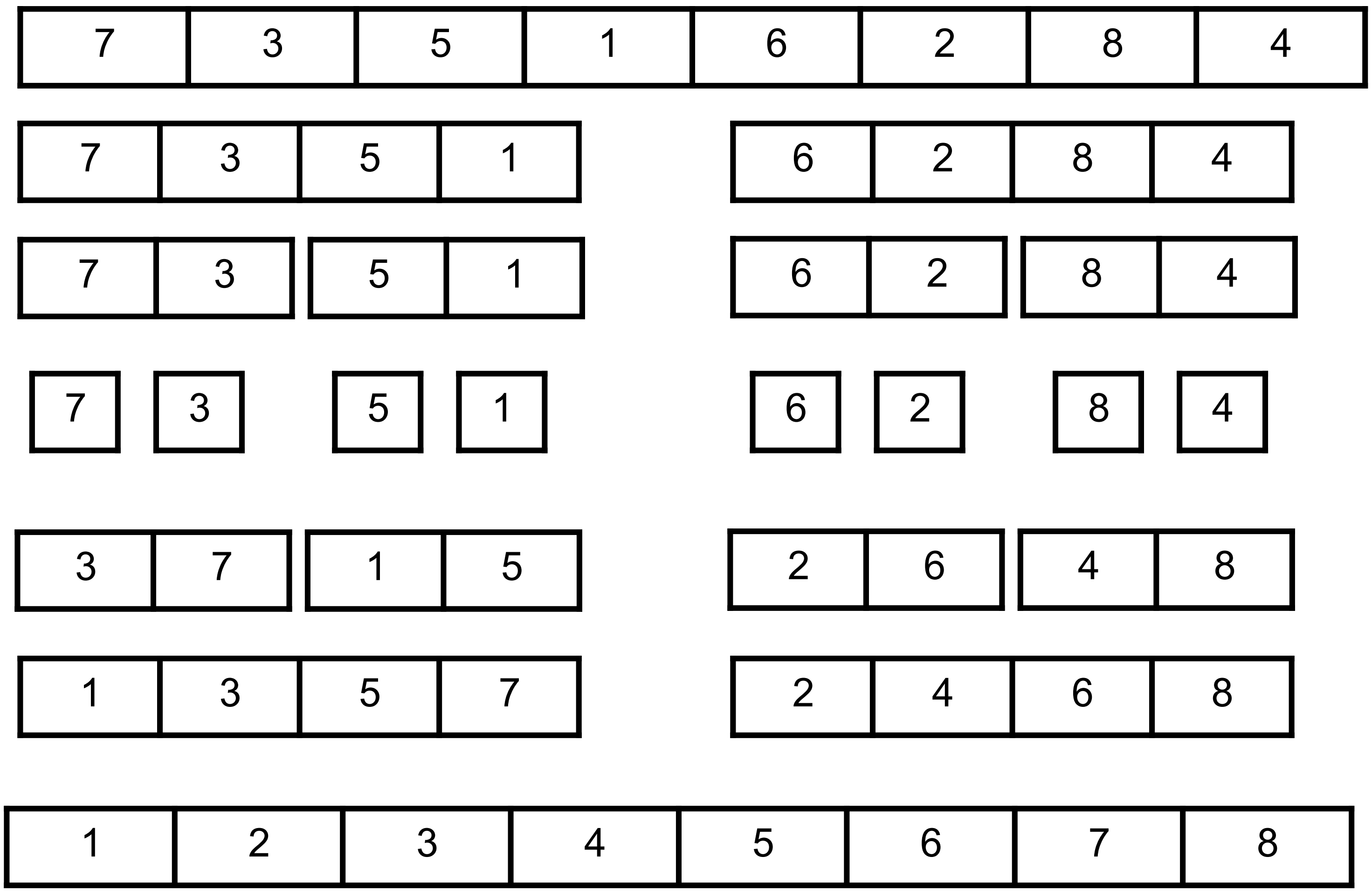
- Iterative approach
  - Use loops to repeat a task, “while”, “for”, ...

- Recursive approach
  - A method calls itself
  - Each time with different parameters
  - Until we reach a trivial base case

Merge Sort

- Basic idea:
  - Recursively divide the array into halves
  - Merge the sorted sub-arrays
- One of the best-known examples of the utility of the divide-and-conquer paradigm.
- Often implemented using a top-down approach and using recursion.
- More efficient than the elementary sorting algorithms we have learned (selection, insertion and bubble)

Merge Sort


7 3 5 1 6 2 8 4
7 3 5 1
6 2 8 4
7 3
5 1
6 2
8 4
7
3
5
1
6
2
8
4
3 7
1 5
2 6
4 8
1 3 5 7
2 4 6 8
1 2 3 4 5 6 7 8

Merge Sort

- Recursive algorithm

```
public void mergeSort(int[] array, int lower, int upper)
{
    if (lower < upper)
    {
        int mid=(lower+upper)/2;
        mergeSort(array, lower, mid);
        mergeSort(array, mid+1, upper);
        merge(array, lower, mid, upper);
    }
}
```

Merging Two Sorted Arrays

- Needs an additional array to hold the results
  - Not in-place

2	7	8	10

1	3	5	9

1	2	3	5	7	8	9	10

Merging Two Sorted Arrays

```
private void merge(long[] workSpace, int lowPtr, int highPtr, int upperBound)
{
    int j = 0;
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1; // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
        if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
        else
            workSpace[j++] = theArray[highPtr++];

    while(lowPtr <= mid)
        workSpace[j++] = theArray[lowPtr++];

    while(highPtr <= upperBound)
        workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
        theArray[lowerBound+j] = workSpace[j];
}
```

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 289

Merge Sort:

```
            sort(a, 0, 15)
sort left     sort(a, 0, 7)
half            sort(a, 0, 3)
                  sort(a, 0, 1)
                    merge(a, 0, 0, 1)
                  sort(a, 2, 3)
                    merge(a, 2, 2, 3)
                  merge(a, 0, 1, 3)
                sort(a, 4, 7)
                  sort(a, 4, 5)
                    merge(a, 4, 4, 5)
                  sort(a, 6, 7)
                    merge(a, 6, 6, 7)
                  merge(a, 4, 5, 7)
                merge(a, 0, 3, 7)
sort right    sort(a, 8, 15)
half            sort(a, 8, 11)
                  sort(a, 8, 9)
                    merge(a, 8, 8, 9)
                  sort(a, 10, 11)
                    merge(a, 10, 10, 11)
                  merge(a, 8, 9, 11)
                sort(a, 12, 15)
                  sort(a, 12, 13)
                    merge(a, 12, 12, 13)
                  sort(a, 14, 15)
                    merge(a, 14, 14,15)
                  merge(a, 12, 13, 15)
                merge(a, 8, 11, 15)
merge         merge(a, 0, 7, 15)
results
```

a[]

merge(a, lo, mid, hi)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Efficiency of Merge Sort

- What is the time complexity? O(nlogn)
  - How many times do we call merge()?
    - It relates to the number of levels in recursion
    - O(log n) times
  - What is the complexity of merge algorithm.
    - O(n)

- Additional space needed
  - Not in-place

Complexity Analysis Using Recursive relation

- Write a recursive relation for time complexity, and solve it to get the big O complexity.
- Recursive relation for merge sort:
  - $T(n)=2*T(n/2)+O(n)$, and we have $n = 2^k$
  - $T(2^k)=2T(2^{k-1})+2^k$
  - $T(2^k)/2^k=T(2^{k-1})/2^{k-1}+1$
  - $=T(2^{k-2})/2^{k-2}+1+1$
  - …….
  - $=T(2^0)/2^0+k$

So: $T(2^k)=k2^k=$nlogn

Complexity Analysis Using Recursive relation

- Write a recursive relation for time complexity, and solve it to get the big O complexity.

- How about binary search
  - $T(n)=T(n/2)+O(1)$

- Linear Search:
  - $T(n)=T(n-1)+O(1)$

Merge Sort:

- Property:
  - Pros: it guarantees to sort any array of n elements in time proportional to nlogn (linearithmic); better than Selection sort, and the average and worst case for Insertion sort.
  - Cons: often need to use extra memory, not in-place.

Quick Sort

- Basic idea:
  - Partitioning (rearrange) the array into two subarrays, and then sort each subarray independently and recursively.
  - When the two subarrays are in-sort, the entire array is sorted.
  - Different from Merge sort where the array is divided in half, in quick sort, the position of the partition depends on the contents of the array.

Quick Sort

- Property:
  - Also follow the divide-and-conquer paradigm.
  - Often use recursion
  - An in-place algorithm
  - Running time is nlogn on the average for an array of length n.
  - Popular, and more often than other sorting algorithms.
  - It is Quick, quicker than other sorting algorithms in typical applications.

Quicksort Algorithm

Given an array of *n* elements:

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Recursively Quicksort two sub-arrays
  - Return results

Quick Sort

- The recursive algorithm

```
public void quickSort(int arr[], int low, int high)
    {
        if (low < high)
        {
            //partition
            int index = partition(arr, low, high);

            // Recursively sort each partition
            quicksort (arr, low, index-1);
            quicksort (arr, index+1, high);
        }
    }
```

Quick Sort

- Two key processes:
  - How to pick a pivot?
    - There are many different ways to choose a pivot, and often we choose to use the first element in the array (subarray).
      - Can always choose the first one
      - Can always choose the last one
      - Choose a median one
  - How to do partition?
    - Goal: partition elements into two sub-arrays:
      - Elements less than or equal to pivot
      - Elements greater than pivot
    - How to do it?

Quick sort: partition code

```
private static int partition(Comparable[] a, int lo, int hi)
{  // Partition into a[lo..i-1], a[i], a[i+1..hi].
   int i = lo, j = hi+1;             // left and right scan indices
   Comparable v = a[lo];             // partitioning item
   while (true)
   {  // Scan right, scan left, check for scan complete, and exchange.
      while (less(a[++i], v)) if (i == hi) break;
      while (less(v, a[--j])) if (j == lo) break;
      if (i >= j) break;
      exch(a, i, j);
   }
   exch(a, lo, j);          // Put v = a[j] into position
   return j;                // with a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (left > right)
- swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

Quick Sort: partitioning

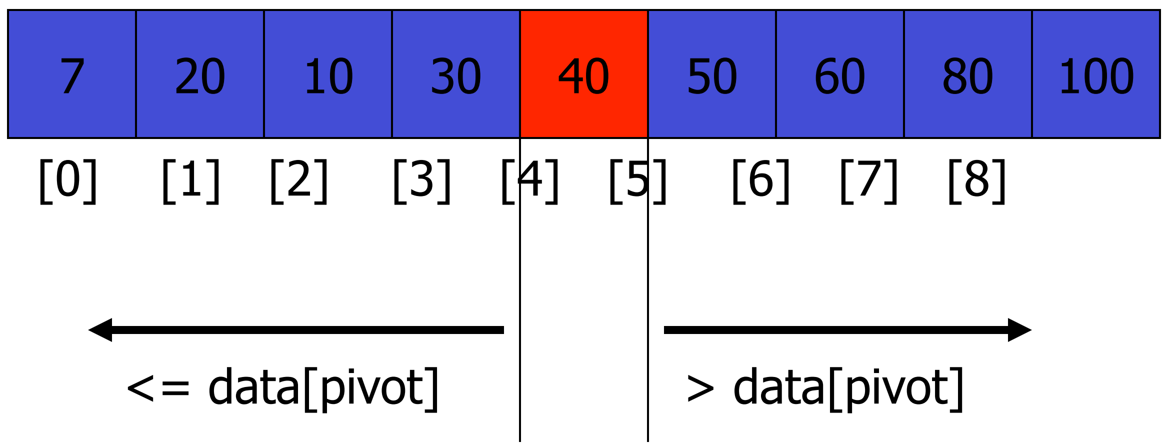
			v							a[]								
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Example

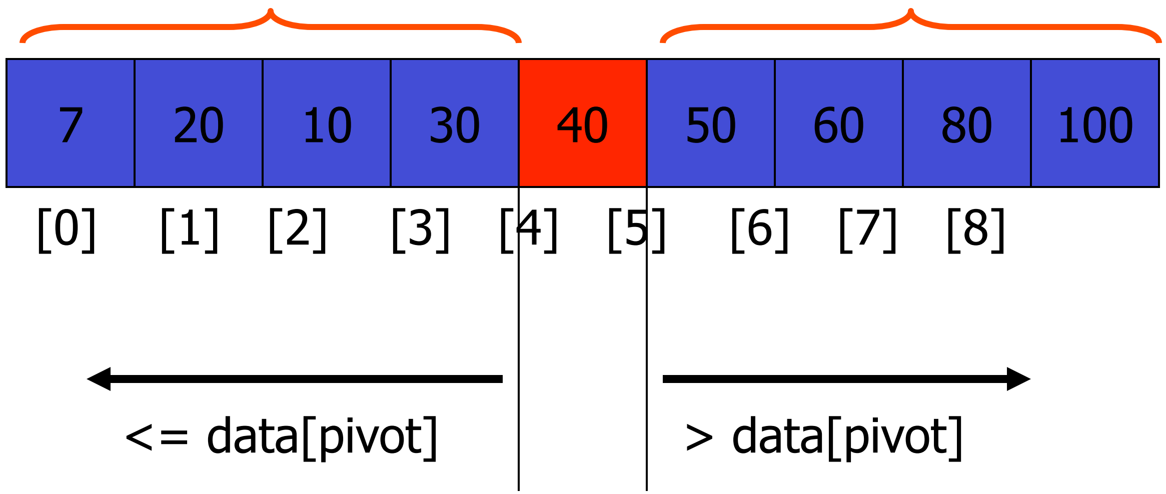
We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Partition Result

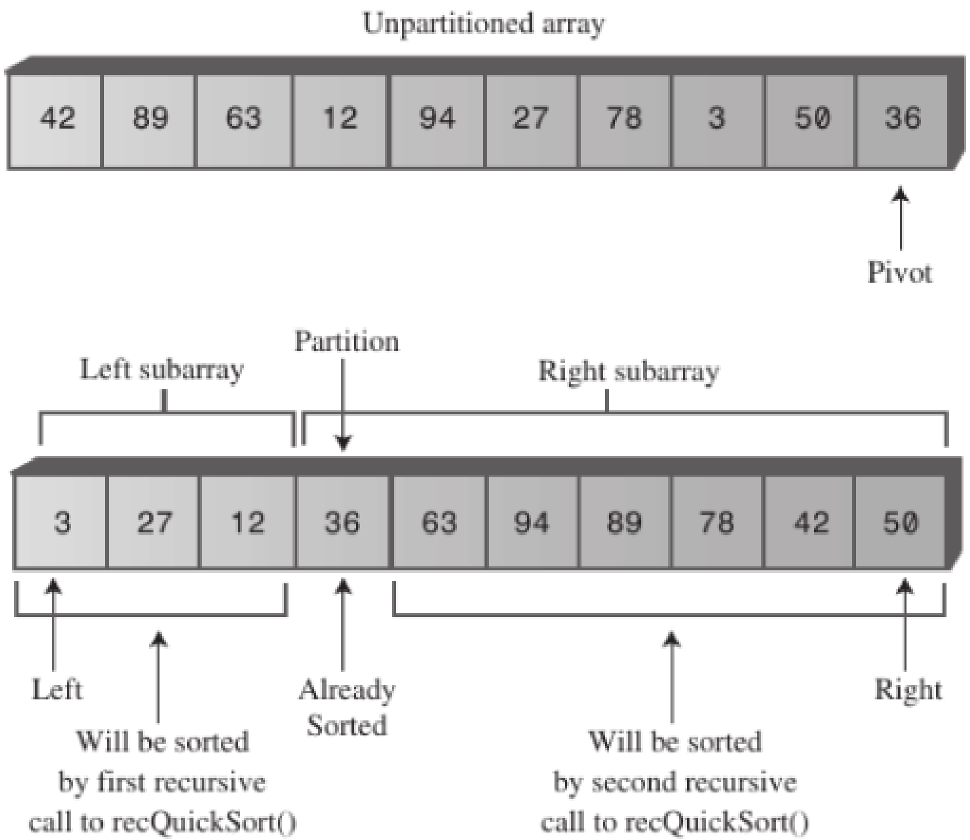


Recursion: Quicksort Sub-arrays



Partitioning the Array (Choosing the last one as pivot)

- Using the last element as the pivot



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 334

Partition(last element as pivot)

```
public int partition(int arr[], int low, int high)
  {
    int pivot = arr[high];
    int i = (low-1); // index determining the end of left partition
    for (int j=low; j<high; j++)
    {
      if (arr[j] <= pivot) // arr[j] should be in the left partition
      {
        i++;
        // swap arr[i] and arr[j]
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }

    // put the pivot in its place
    int temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;

    return i+1;
  }
```

What is the time complexity of partition method?

O(n)

Choosing Pivot

- For simplicity we used the last element as the pivot.
  - fine if the input is random
  - What happens if the array is sorted?
    - We will have a subarray of size 0 and a subarray of size n-1
    - n recursion levels → O(n^2) total complexity

- Ideally we want to divide the array into halves
  - It makes log (n) recursion levels→ O(n log n) total complexity

Quick Sort: Improvement

- Picking a better pivot
  - rather than picking the first/last as pivot:
    - drawback: if the array is already sorted, the partition will be biased to worst case: $\sim n^2$
    - Median of three elements:
    - It will be the best if we can find the median of the entire array, so to evenly partition into two halves
    - Unrealistic, so find the median of three elements for any subarray:
    - The first, middle and the last one
    - Pivot = median of a[0], a[n/2] and a[n-1]
    - a[0]=8, a[n/2]=3, a[n-1]=5, then the median is 5.

Partition(general, pivot isn't the last)

```
public int partition(int[] arr, int left, int right)
{
    int pivot = medianOfThree (arr[], left, right); // Pick pivot point

    while (left<= right) {

      while (arr[left] < pivot)
          left++;

      while (arr[right] > pivot)
         right--;

     // Swap elements, and update left and right
     if (left<= right) {
         int temp= arr[left];
         arr[left]= arr[right];
         arr[right]=temp;
         left++;
         right--;
      }
  }
 return left;
}
```

You can find median of three with comparisons.
It is easier to put the median of the three at the end and use the previous partition code.

Quick Sort: performance analysis

- Best case: partition into evenly half
  - $T(N)=T(N/2)+N$ (n is for the compares in partitioning)
  - proved to be NlogN (Merge sort)

- Worst case: two partitions are extremely biased, i.e. left subarray contains only 1 element, and right one contains the N-1
  - $T(N)=N+N-1+N-2+\ldots\ldots.1=N(N+1)/2 \sim N^2$

Quick Sort: performance analysis

- Average Case: (can be obtained by shuffling)

1. $T_N = N+1+(T_0+T_1+T_2 \ldots + T_{N-2}+T_{N-1})/N+(T_{N-1}+T_{N-2}+ \ldots . T_0)/N$
2. $NT_N = N(N+1)+2(T_0+T_1+\ldots .. T_{N-2}+T_{N-1})$
3. $(N-1)T_{N-1} = (N-1)N+2(T_0+T_1+ \ldots . T_{N-3}+T_{N-2})$

Subtract 2-3:

4. $NT_N-(N-1)T_{N-1} = 2N+2T_{N-1}$
5. $T_N/(N+1) = T_{N-1}/N + 2/(N+1)$

$$=T_{N-2}/(N-1)+2/N+2/(N+1)$$

$$=T_{N-3}/(N-2)+2/(N-1)+2/N+2/(N+1)$$

$$= \ldots \ldots \ldots$$

$$=T_1/2+2(1/3+1/4+1/5+ \ldots .. 1/(N+1)$$

6. $T_N \sim 2(N+1)(1/3+1/4+1/5+ \ldots . 1/(N+1)) \sim 2N\log N$

the integral of function 1/x is logx